Vol.2013-SE-182 No.23
2013/10/25

センサデータサーバのための分散データベースの構築・運用に関する考察

永井 琢也[1]　満田 成紀[2]　福安 直樹[2]　松延 拓生[2]　鯵坂 恒夫[2]

概要：センサデータサーバとは，センサデータを扱うデータベースサーバである．本研究の目標は，センサデータサーバを分散環境で構築することである．これはセンサデータサーバのスケーラビリティや可用性を高めるためである．目標に向けて，センサデータサーバを分散 RDB と分散 KVS で仮想環境上に試作した．試作したものを，構築時のコストやデータの保存方式などの構築面と処理速度，スケーラビリティやネットワーク障害時などの運用面から考察した．考察の結果から，それぞれの環境の特徴を抽出した．

1. はじめに

近年センサを用いて実世界の様々な環境や状況を観測し，その結果を利用するアプリケーションが多く存在する [1]．センサが計測する情報の種類や形態も様々である．

センサデータサーバとはセンサデータを保存・管理するためのものである．センサから送られてくる生データはアプリケーションはそのまま使用しにくいため，加工する必要がある．センサデータサーバが生データを加工して保存することで，複数のセンサデータを組み合わせて利用したアプリケーションが容易に実装できる．しかし，センサデータのように増え続ける大量のデータはビックデータと呼ばれ，一台のマシンで処理が困難な状況である．このような問題の解決策に処理やデータを分散する方法がある．

ビッグデータを分散して処理するための技術として NoSQL が注目を集めている．本研究で使用する Riak も NoSQL の一つである．Riak などの NoSQL の一部は KVS と呼ばれるシンプルなデータモデルでデータを管理する．KVS はデータ間に厳しい制約がないため，複数のサーバで分散環境を構築することが容易である．NoSQL には様々な製品が含まれており，それぞれに特長を持っている．

本稿では，構築や運用を考えた時にセンサデータサーバはどのような分散環境が適しているか考察する．そのために，従来のセンサデータサーバを参考に MySQL を用いた分散 RDB と Riak を用いた分散 KVS を比較する．

[1] 和歌山大学大学院システム工学研究科
Wakayama Univ., Fucalty of Systems Engineering
[2] 和歌山大学システム工学部
Wakayama Univ., Fucalty of Systems Engineering

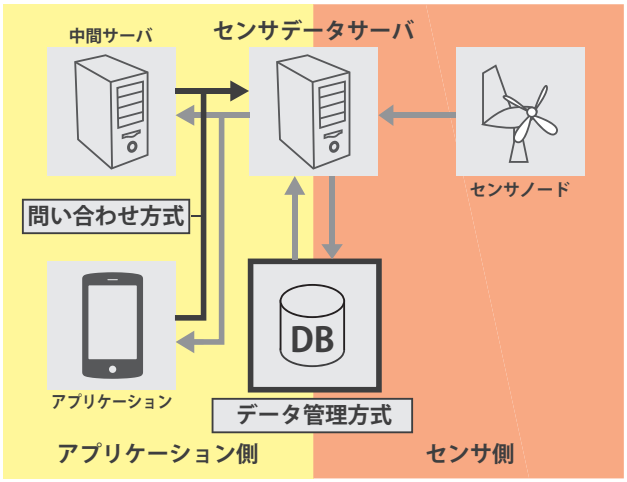

図 1　センサデータサーバの役割
Fig. 1　The Role of Sensor Data Server

2. センサデータサーバとその構築環境

2.1 センサデータサーバ

センサデータサーバとは，大きく分けて次の 2 つの機能を持つサーバである．図 1 にセンサデータサーバの役割を示す．1 つは，センサノードからセンサ情報を受け取り，そのセンサ情報を一定の規則に基づいて変換，保存するセンサ側の機能である．もう 1 つは，アプリケーションまたは他の中間サーバからの要求に対して適切なセンサデータを選択，出力するアプリケーション側の機能である．これらの機能により，センサデータは加工された後保存され，アプリケーションは生データの加工などを気にせずにセンサデータを使用することが可能である．本稿では，センサデータサーバを分散環境で構築するための比較・考察を

行う．

2.2 目標とする分散環境

本研究で扱うセンサデータサーバは，センサの種類が増えた時，パフォーマンスやディスク容量が必要な時やネットワークに障害が発生した時に対処が容易であるような分散環境の構築を目標としている．

2.2.1 センサデータサーバに求める特性

センサデータサーバを運用していく上で，扱うセンサの種類が増えることが想定される．センサの種類が増えた時に発生する問題は，保存するセンサデータの形式である．センサごとに固有の形式を持つため，センサの種類が増えた場合，新たなデータ形式でセンサデータが送られてくる．センサデータサーバはこの新たなデータ形式に対応する必要がある．

スケーラビリティとは，システムを運用管理していく上で遭遇する問題に対して対策の取りやすさである．センサデータサーバを運用していく上で問題となるのは，データが増え続けていく点である．データが増え続けることによる問題は，データを取得する際のパフォーマンスの低下とディスク容量の不足である．データが増えれば増えるほど大量のデータを検索処理することになり，期待するデータを受け取るまでの時間が長くなることが考えられる．また，運用を続ければ増え続けるデータはディスク容量を圧迫する．そのためセンサデータサーバにはスケーラビリティが必要である．

一部のネットワークが何らかの障害で分断されてしまってもシステム全体は動き続ける必要がある．システム全体がいつでも利用できる状態を可用性が高いと表現する．センサデータサーバはたくさんのセンサノードからデータを受信する．センサデータを随時保存することで統計処理などが可能となる．従って，センサデータサーバには可用性が必要である．

2.2.2 スケールアウト

これらの問題の対策としてスケールアウトとスケールアップが考えられる．スケールアウトとは汎用的なサーバを複数台増やしていくことでサーバ群のパフォーマンスや容量を向上させる方法である．サーバ数を増やすことでディスク容量が増える．また，サーバ数が増えることで処理をより多く分散させることが可能となりパフォーマンスが向上する．スケールアウトを容易に行えることはスケーラビリティの向上にもつながる．また，一台のサーバがダウンまたはネットワークが分断されたとしても他のサーバが機能を補うことができるため，システム全体として稼働することが可能である．このためシステム全体の可用性も高くすることが可能である．スケールアップとは，サーバをより高パフォーマンスで容量の大きいものと交換する方法である．しかし，本研究ではセンサデータサーバにスケールアップは適用しない．スケールアップはパフォーマンスや容量を向上させることは可能だが，一台のマシンでの運用のままでは限界がある．また，一台のマシンで高可用性を実現するのは困難である [2]．よってセンサデータサーバにはスケールアップは不適である．また，スケールアップはデータの整合性が重視される場合などの運用性向上に適用される方法である．本研究で扱うセンサデータサーバは一つ一つのデータの厳密な整合性は必要としない．それはセンサデータ一つ一つにはあまり意味がなく，統計処理など複数のデータを処理して使用する場合がほとんどだからである．センサデータサーバのスケーラビリティ，可用性の向上のための方法としてスケールアウトを選択する．

3. 分散 RDB を用いた構築

3.1 RDB によるデータ管理方式

設計したセンサデータサーバについて説明する．適切なセンサデータサーバの実装を行うためには，データ管理方式と問い合わせ方式が整合している必要がある．そのため，設計は問い合わせ方式，データ管理方式の順に行う．データ管理方式を先に設計してしまうと，問い合わせ方式を設計する際に設計できる問い合わせの種類に制限ができてしまうからである．また，センサデータサーバは誰もが自由に利用できるオープンな仕組みであることを目指した．そのため問い合わせおよびデータ管理方式は，あらゆるアプリケーションに対応し，要求を効率よく処理できるよう設計する．

3.1.1 問い合わせ方式

設計した問い合わせ方式を以下に示す．

検索機能

- センサタイプによる検索
- センサの座標による検索
- センサデータが観測された時間による検索
- センサが固有に設定されているセンサ ID による検索

センサデータ処理機能

- 要素に対しての四則演算機能
- 検索結果の最大値と最少値を出力する機能
- 検索結果の数を数える機能

抽出された機能として検索機能 4 つとセンサデータ処理機能 3 つを設計する．これらの機能を組み合わせてアプリケーションなどは期待するセンサデータを受け取ることが可能である．これらの機能は対象としたセンサデータに，時間情報と位置情報が必ず含まれることを前提としている．

センサタイプによる検索

指定されたセンサタイプのセンサデータのみを検索する．センサは複数指定することができる．

センサの座標による検索

センサデータに含まれている位置情報を用いた検索で

表 1　共通部分テーブル
Table 1　Common Data Table

カラム	id	time	x	y	height	sensor_id	sensor_type	hash
型	int	timestamp	double	double	double	int	text	int

表 2　センサ固有部分テーブル
Table 2　Sensor Dependent Data Table

カラム	id	common_id	sensor_name	unit	hash	c1	c2	...	cn
型	int	int	text	text	int	-	-	...	-

各センサ種ごとの固有情報

ある．座標検索には以下の 3 つの種類がある．

- 点検索
  座標を一点指定して，その座標にあるセンサデータを取得する．
- 長方形領域検索
  長方形領域の対角の 2 点を指定することで，その長方形領域内にあるセンサデータを取得する．
- 円形領域検索
  中心点と円の半径を指定することで，円形領域内にあるセンサデータを取得する．

時間による検索

センサデータに含まれる時間情報を用いた検索である．時間による検索は一点を指定して検索する点検索と観測時間に幅を持たせて検索する幅検索の 2 つがある．

センサ ID による検索

センサ ID とはセンサに割り振られている固有の ID のことである．センサタイプが同じでもセンサ ID を用いれば個々のセンサを識別することができる．

3.1.2　データ管理方式

データ管理方式は設計した問い合わせ方式を効率良く処理できることを目指して設計する．設計したデータ管理方式を表 1，表 2 に示す．センサデータサーバにおけるデータ管理方式は様々なセンサから送られてくるセンサデータに対応する必要がある．そのためセンサデータすべてに共通している共通部分と，各センサの種類に固有の固有部分に分け，それぞれを共通部分テーブルと固有部分テーブルに保存する．テーブルを分割し，共通の要素を抽出することで，様々なセンサ情報を対象とした検索を可能にする．

センサデータサーバは大量のセンサデータを扱う．このことを踏まえ検索効率の向上を目的に，テーブルにインデックスを作成する．センサデータを選択する際に，データに含まれる時間と場所による検索が頻繁に行われる．センサデータには時間情報と座標情報が必ず含まれているからである．このことから，共通部分テーブルの time カラムと point カラムの組み合わせにインデックスを作成する．また，共通部分テーブルと固有部分テーブルを連結させる際に，固有部分テーブルの common_id カラムを使用するため common_id カラムにもインデックスを作成している．これらのインデックスにより検索効率が向上した．

3.2　ストレージエンジン SPIDER による分散化

分散環境を構築するために使用するストレージエンジン「SPIDER」について説明する．SPIDER とは MySQL 用のストレージエンジンである [*1]．RDB を用いてデータシャーディングを実装することが可能である．データシャーデングとはデータを分割し，それらを別々のノードに格納することでシステムのパフォーマンスを向上させる方法である．

RDB のパフォーマンスを向上させる場合レプリケーションによるスケールアウトが一般的である．レプリケーションとは元となるサーバ（マスタ）の複製（スレーブ）を複数用意し，負荷を分散させる方法である．データが複数のサーバに分散するため，高い可用性を実現できる．しかし，データはマスタと同じものが複製されるので増え続けるデータには不向きである．また，スレーブを複数用意することで分散できる処理は読み出し処理だけである．書き込みや更新はすべてマスタに集中するため，センサデータサーバのようにセンサノードからの書き込み処理の多い状況には適さない．これらの理由からレプリケーションではなく SPIDER を用いたデータシャーディングを選択する．

図 2 は SPIDER ストレージエンジンを利用して実際に構築したセンサデータサーバの構成である．ノード 1 に SPIDER ストレージエンジン，ノード 2 から 4 に InnoDB を利用したテーブルを構築する．ノード 1 は実データを持たず，他の MySQL サーバへのリンクを保持する．リクエストはノード 1 に送られてくるが，実際の処理はノード 2 から 4 で行われる．これにより書き込み性能を分散することを可能としている．

4.　分散 KVS を用いた構築

4.1　Riak

本研究では分散 KVS の構築に Riak を使用する．Riak とは，Basho Technologies [*2] が開発しているオープンソースデータベースである．

4.1.1　KVS

NoSQL におけるデータモデルの一種である KVS について説明する．KVS は非常にシンプルなデータモデルで，データはキーとバリューの組み合わせで一つの塊として保存される．データを呼び出す際は，キーを指定することで，そのキーに紐づいているバリューを取り出すことができる．Riak のデータモデルも KVS である．

リレーショナル型のデータモデルと大きく異なる点は，データを登録するためのスキーマを設定しなくてよいこと

[*1] http://spiderformysql.com/
[*2] http://basho.com/riak/

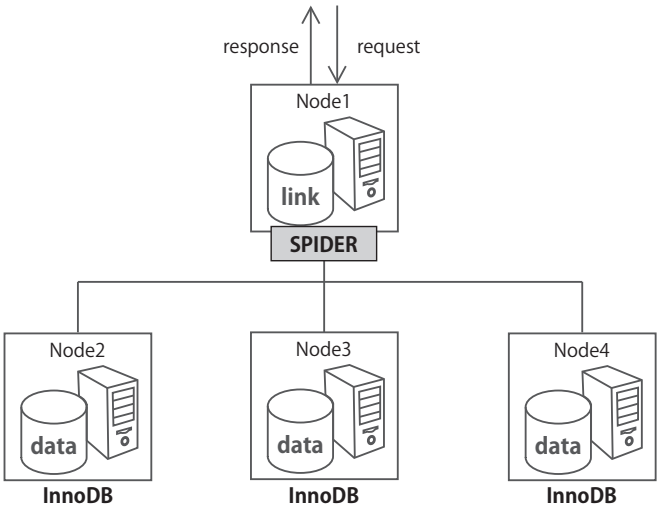

図 2 SPIDER を用いた分散 RDB
Fig. 2 Distributed RDB with SPIDER

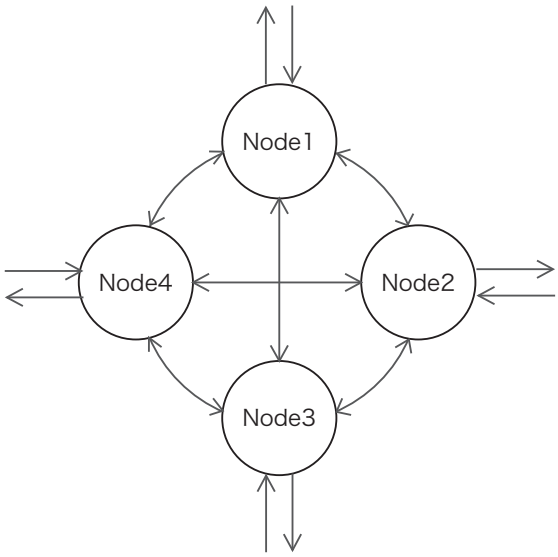

図 3 Riak のアーキテクチャ
Fig. 3 The Architecture of Riak

である．スキーマとはデータベースの構造であり，各データの型やデータの大きさなどを定めるものである．リレーショナル型のデータモデルでは，あらかじめスキーマを設定しておくため，データをすべて予測しておく必要がある．KVS では，データ（バリュー）にはキーが紐づいており，バリューの中身は意識しないので，いろいろなデータの型に柔軟に対応可能である．一方で，シンプルなデータモデルであることにより，データ間の厳密な整合性を保つことはできない．整合性に関して Riak では結果整合性を採用している．結果整合性とは，更新処理が始まった瞬間のあとわずかな時間を経たのちに整合性が確保されるものである．一瞬整合性は崩れるが，結果的に整合性が保たれる．これは厳密な整合性を維持しないことで高パフォーマンスを実現するためである．常に整合性のとれたデータを参照可能であることを保証する RDB と違い, ある瞬間には更新前のデータが読みだされる可能性がある．しかし厳密な整合性を保つためにはトランザクションを排他制御する必要があり，その分パフォーマンスは低下する．センサデータサーバは多種多様なセンサデータを対象とし厳密な整合性は必要ないため，KVS に適している．

また, Riak にはキーとバリューに加えて, キーとバリューの組み合わせをグループ化するバケットが実装されている．バケットは RDB におけるテーブルに相当する．これによりバケットとキーの組み合わせがバリューと対になるデータモデルが実現可能である．本研究ではセンサの種類ごとにバケットを用意した．

4.1.2 P2P 型アーキテクチャ

Riak はマスターレスなアーキテクチャである．ノード間を P2P 接続することでお互いの情報を交換するアーキテクチャである．図 3 に Riak のアーキテクチャを示す．仮にノードが一つダウンしても他のノードがその役割を引く継ぐことが可能である．そのため，単一障害点がない特徴がある [3]．これにより高可用性が期待できる．

Riak のアーキテクチャを実現するためにゴシッププロトコルやコンシステント・ハッシングがある．ゴシッププロトコルとはデータの割り当てや配置に関する情報を通信するプロトコルである．このプロトコルを使用することで，クラスタに属するノードの状態が把握することが可能である．各ノードがお互いの状態を監視し続けることで，ノードがクラスタから分断されたことや，復帰してきたことを常に把握することができる．これは本来クラスタの状態を確認するマスタの役割を代替している．コンシステント・ハッシングとは分散システムにおいて各ノードに均等にデータを振り分けるためのアルゴリズムである．コンシステント・ハッシングは分散システムの各ノードを理論的にリング状に配置する．各ノードにはハッシュ値が指定され，次のノードまでのハッシュ値を担当する．データを書き込む際にはそのデータのキーからハッシュ値を計算し，ハッシュ値が当てはまるノードにデータが書き込まれる．コンシステント・ハッシングのメリットはデータを分散できることに加えて，ノードを追加，削除する場合のシステム全体の負荷が少ないことが挙げられる．例えば新たにノードを追加する場合は，既存のノード間に配置されるようハッシュ値を割り当てるだけでノードの追加を行うことが可能である．このようにシステム全体を変更することなくノードの追加を行うことが可能である．Riak ではノードが追加された場合，新ノードが担当する部分にそれまで担当していたノードからデータの複製を自動的に行う．

4.1.3 MapReduce

KVS はシンプルな構造であるため，それだけではキーに対してバリューを取得することしかできない．しかしセンサデータを検索する場合この機能だけでは不十分である．インデックスや MapReduce などを用いることである

程度複雑な検索が可能である．MapReduce とは，大規模なデータを複数のノードで並列分散処理するためのプログラミング・パターンである [4]．Map 処理で膨大な元データを分解して必要な情報を抽出する．抽出したデータを有用な形へ変換して出力する．Reduce 処理では Map 処理によって抽出されたデータをひとまとまりにして出力する．Map 処理を複数のサーバに分散することで，処理対象が膨大であっても現実的な時間で処理することができる．ただし，検索が複雑になるほど実行時間が必要である．それは，MapReduce によって分散される一つ一つの Map タスクが単純な処理であることが前提であるからである．

4.2 Riak を用いたセンサデータ管理方式

Riak を用いて実際に構築したセンサデータサーバについて説明する．4.1.1 節で述べたように，KVS はシンプルなデータモデルであるため，それだけではキーに対してバリューを取得することしかできない．座標による範囲検索など複雑な検索を行うために，キーとインデックスを設計した．キーとインデックスの設計により，時間幅検索や座標領域検索の実装を目指す．

4.2.1 キー設計

キーは時間，経度，緯度を”_”で繋げたものにした．キーの例を以下に示す．

20130501000000_135.000001_34.000001

キーにこれらの情報を含ませたのは，キーフィルタリングを利用するためである．キーフィルタリングとは Riak が実装しているデータ検索方法の一つである．キーに含まれる文字列をフィルタリングすることで幅検索や AND 検索など様々な検索を行うことができる．ただし，実行速度が遅いという問題点がある．キーフィルタリングを行う際，全データのキーを参照していると考えられるからである．

4.2.2 インデックス設計

Riak ではキーのほかにセカンダリインデックスを設定することができる．セカンダリインデックスとは，ユーザが独自に設定できるインデックスである．インデックスとして時間，経度，緯度の 3 つを設定した．こうすることで時間検索や座標検索が効率よく実行可能となる．インデックス検索は 4.2.1 節で述べたキーフィルタリングと比べかなり高速である．ただし，キーフィルタリングのような柔軟な検索はできない．インデックスを用いた検索で可能なのは，インデックスの値が一致するバリューの取得と，インデックスの値を用いた幅検索である．インデックスを用いた検索では AND 検索は実装されておらず，検索の組み合わせは行えない．このように設計したデータ管理方式で RDB とのパフォーマンスの比較を行う．

4.2.3 クエリの実装方法

クエリを実装するためにセカンダリインデックスを利用することで時間や座標を絞り込んだ検索が可能である．この検索によって取得したキーの集合に対して MapReduce を用いてバリューを取得する．図 4，図 5，図 6 に今回使用したクエリの実装を示す．それぞれ時間幅を用いた検索，長方形領域を用いた検索，時間幅と長方形領域を用いた検索である．長方形領域を用いた検索を行う場合，x 座標と y 座標の 2 つの指標を検索する必要がある．セカンダリインデックスはアンド検索が実装されていないため，キーフィルタで実装する．

```
# $st, $et に開始時刻，終了時刻が入る
$result = $client
  ->add("gps")
  ->indexSearch("time", "int", $st, $et)
  ->map(array("riak_kv_mapreduce",
                 "map_object_value"))
  ->reduce("Riak.reduceSort")
  ->run();
```

図 4 時間幅を用いた検索の実装
Fig. 4 A Query for a Specific Time Range

```
# $miny, $maxy, $minx, $maxx に緯度経度の開始座標，終了座標が入る
$result = $client
  ->add("gps")
  ->key_filter(array('tokenize', '_', 3),
   array('between', $miny, $maxy))
  ->key_filter_and(array('tokenize', '_', 2),
   array('between', $minx, maxx))
  ->map("function (v) { return [v]; }")
  ->reduce("Riak.reduceSort")
  ->run();
```

図 5 長方形領域を用いた検索の実装
Fig. 5 A Query for a Specific Rectanangle Area

5. 比較と考察

この章では実装した分散 RDB と分散 KVS について比較と考察を行う．正常にセンサデータサーバが稼働している時のパフォーマンスについて比較する．センサ特有のデータ形式への対応を考え，扱うセンサの種類が増えた場合について比較する．センサデータサーバを運用していく上でスケールアウトする場合を想定し，サーバの追加しやすさや追加した時のデータの流れに着目する．ネットワークに障害が発生し，一部のノードがシステムから独立してしまう可能性がある．この時，システム全体や独立したノードがどのように振る舞うのか比較する．最後にネットワーク障害が復旧し，独立していたノードがシステムへ復旧してくることを想定する．この時に，データがどのような状態で復旧するか比較する．

```
# $st, $et に開始時刻，終了時刻が入る
# $miny, $maxy, $minx, $maxx に緯度経度の開始座標，終了座標が入る
$result = $client
->add("gps")
->indexSearch("time", "int",  $ss, $es)
->map("function (v) {
var data = Riak.mapValuesJson(v)[0];
if ($minx <= data.x && data.x <= $maxx
&& $miny <= data.y && data.y <= $maxy) {
return [v];
} else {
return [];
}
}")
->reduce("Riak.reduceSort")
->run();
```

図 6 時間幅と長方形領域を用いた検索の実装
Fig. 6 A Query for a Specific Time Range and Rectanangle Area

表 3 仮想環境を構築したマシンのスペック
Table 3 Spec of Virtual Environment

OS	XenServer
CPU	Intel(R) Core(TM) i7-3770 CPU @ 3.4GHz
メモリ	16GB

表 4 各分散環境で使用した製品
Table 4 Using Products for Distributed Database

分散 RDB	DB	MySQL 5.5.14
	ストレージエンジン	spider 2.26 for MySQL 5.5.14
分散 KVS	DB	Riak1.4.2

5.1 正常時のパフォーマンス

分散システムが正常に稼働してる状態でのパフォーマンスについて比較，考察を行う．仮想環境を構築したマシンのスペックを表 3 に示す．パフォーマンステストは分散 RDB と分散 KVS をそれぞれ表 2，表 3 に示す環境を仮想環境の上に構築した．仮想化には XenServer [*3] を利用している．仮想環境は表 3 上に構築した．それぞれの分散環境はノード数 4 つで構成した．一つのノードには CPU2 コア，メモリ 3.0GB を割り当てた．各環境で用いた製品を表 4 に示す．

パフォーマンステストはデータ数 50 万件でから 500 件を絞り込む検索処理の実行時間を計測する．検索は以下の 3 種類の条件で行った．

(1) 時間幅による検索

[*3] http://www.citrix.co.jp/products/xenserver/xenserver.html

表 5 パフォーマンステスト結果
Table 5 A Result of Performance Test

条件	(1)	(2)	(3)
分散 RDB	1.97	1.96	1.99
分散 KVS	0.36	5.59	3.91

(2) 長方形領域による検索

(3) 時間幅と長方形領域による検索

これらの検索条件でデータを取得する PHP スクリプトを Riak,RDB 両方のものを作成し，実行時間を計測した．実行計測には PHP スクリプト内で処理の初めと終わりの時刻を取得し，その差分を取ることで計測した．

対象センサに gps を想定した．テストデータはテストデータを自動生成するスクリプトを作成し，スクリプトで生成されたものを利用した．gps のテストデータは座標 (135.0, 34.0) の点から座標 (135.5, 34.5) の点まで 50 万秒かけて移動するテストデータが格納されている．パフォーマンステストの結果を表 5 に示す．それぞれの値は 5 回計測したものの平均（秒）である．

分散 RDB はどの条件においても安定した実行時間であった．一方で分散 KVS は実行時間にばらつきが見られた．これは KVS というデータモデルが関係していると考えられる．KVS はシンプルなデータモデルであるため複雑な検索を得意としない [5]．時間幅検索のような一つの値を参照する検索は分散 RDB と比べて高速に実行できることがわかった．これはセカンダリインデックスを利用できるためである．しかし，長方形領域検索や時間幅と長方形領域を用いた検索では分散 RDB よりも低速な結果となった．長方形領域のような x 座標と y 座標の 2 つの値を参照するアンド検索には時間がかかると考えられるからである．長方形領域検索ではセカンダリインデックスはアンド検索が行えないため, キーフィルタリングを利用して検索したことも実行時間が遅くなった原因の一つである．時間幅と長方形領域での検索が長方形領域のみの検索より少し高速なのは，時間幅検索を行ったあとで長方形領域検索を行っているからである．時間幅でデータを絞った後は，MapReduce に処理を埋め込んで長方形領域を絞り込んだ．セカンダリインデックスとキーフィルタを組み合わせた検索は実装されていなかったからである．また，分散 KVS において MapReduce を用いてそれぞれの検索を行った．しかしノード数が 4 つであるため，処理の分散数が少なく MapReduce の特徴を発揮できていないことも考えられる．このことから今後の課題としてより多くのノード数で実行し，パフォーマンスを計測することや分散 KVS 上で複雑な検索をどのように実現するか検討することが挙げられる．また，実際の稼働環境ではより多くのデータが格納されていると考えられる．より多くのテストデータで実環境を想定したパフォーマンステストを行う必要がある．

表 6　Riak におけるノードの追加コマンド例
Table 6　The Example Commands To Add Node On Riak.

```
# riak-admin cluster join riak@127.0.0.1
# riak-admin cluster plan
# riak-admin cluster commit
```

5.2　スケールアウトを行う場合

スケーラビリティの面からスケールアウトを行う場合について比較，考察を行う．

分散 RDB の場合，SPIDER ノードが分散システム内のノードをすべて把握している必要がある．これは，分散システムを構築するテーブル設計の際にデータの保存先を決定するからである．仮にノードを追加し，SPIDER ノードのパーティションを変更したとする．SPIDER ノードは各ノードへのリンクを保持しているので，リンク先と実際にデータを保存しているノード間の整合性が取れなくなる．この問題を解決するためには，一度全データをダンプしておき，SPIDER ノードのパーティション設定を変更し，その後でダンプしておいたデータをリストアする．こうすることで再度保存先とのリンクが張りなおされ，整合性を保ってノードを追加することができる．ただし，リストアが完全に終了するまでは期待したデータの取得できない可能性がある．また，今回のテストデータ 50 万件を分散 RDB 上に代入するために約 35 時間必要とした．SPIDER を用いた分散 RDB のリストアには長時間必要で，その間パフォーマンスが低下することが考えられる．

分散 KVS の場合，ノードの追加は新たなノードで表 6 に示すコマンドを実行するだけである．Riak はコンシステント・ハッシングによりデータシャーディングを実現しているからである．Riak では何も指定しない場合はすべてのノードがデータを均等に保持するようにノードが追加される．追加されたノードはゴシッププロトコルにより検出され，整合性が保たれるように自動的にデータが複製される．データの複製に掛った時間はデータ数 50 万件の場合約 1 分だった．分散 RDB と比べると低コストでスケールアウトを行うことが可能である．

5.2.1　センサの種類が新たに追加された場合

センサデータサーバがセンサ特有のデータ形式にどのように対応するか比較する．

分散 RDB の場合，スキーマ設計の制限があるため，新たな固有部分テーブルをデータ形式にあわせて作成する必要がある．SPIDER で分散 RDB を構築する場合，すべてのノードに同じテーブルを用意する必要がある．新たに作成する固有部分テーブルはすべてのノードに作成する．

分散 KVS の場合，Key と Value の組み合わせでデータを保存するため，スキーマを意識する必要はない．Riak であれば JSON 形式で Value を保存する．そのためデータ形式に柔軟に対応可能である．

5.3　ネットワークに障害時のシステム全体の動作

ネットワークが分断され，一部のノードが切り離された場合のシステム全体の振る舞いについて比較する．

分散 RDB の場合，分断されたノードが担当していた処理はすべてできなくなる．SPIDER ノードは各ノードへのリンクを保持するだけで，実際の処理はすべてリンク先のノードが行うからである．データもリンク先で保持しているため，参照できない．SPIDER ノードがシステムから独立してしまった場合はシステム全体が機能できなくなる．

分散 KVS の場合，そのまま処理を継続することができる．各ノード同士が P2P で接続されており，複製データがあるからである．ネットワーク障害によってノードがシステムから独立するとゴシッププロトコルによって検出される．Riak はそのノードをコンシステント・ハッシングから自動的に切り離す．切り離した後は，システムが自動的に各ノードのハッシュ値を調節し，データを複製する．分断されたノードが担当していたデータを参照することも可能である．独立したノードが担当していた部分はコンシステント・ハッシングによって新たに割り当てられたノードが担当する．マスターレスなアーキテクチャのため，一定の部分が独立したことでシステム全体が停止することはない．

センサデータサーバに当てはめると，分散 RDB において SPIDER ノードがダウンし，システム全体がダウンすることは避ける必要がある．このためには，SPIDER ノードをレプリケーションするなどして障害性を高める必要がある．一方分散 KVS では，マスタを持たないためシステム全体がダウンする可能性は，分散 RDB に比べ低い．また，分散 RDB ではノードが分断されてしまうとそのノードのデータは参照できなくなる．この問題を解決するためには，レプリケーションしてバックアップを作成しておくことが考えられる．ただし，SPIDER ノードがデータのリンクを保持しているので，ノードが分断された後，適切にバックアップからデータを参照できるかは確認する必要がある．一方で，分散 KVS は常にデータの複製を作成する．このため，ノードが分断されてもデータを参照することができる．システム全体をネットワーク障害の面から見ると，センサデータサーバには分散 KVS が適している．

5.4　システムから独立したノードの振る舞い

ネットワークが分断され，システムから独立したノードについての考察を行う．

分散 RDB の場合，リクエストは必ず SPIDER ノードを経由する必要がある．そのためネットワークから独立したノードはその間リクエストを処理することができない．独立したノードの IP アドレスがわかっている場合はセンサはセンサデータを直接そのノードに保存することができる．ただし，SPIDER ノードを経由しないことでデータへのリ

ンクを生成できないため，システム全体はそのデータを把握できない．独立したノードに直接リクエストを送る場合は，そのノードが保持しているデータは参照可能である．

分散 KVS の場合，マスターレスなアーキテクチャのため全ノードがデータを受信可能である．そのため，たとえ分散システムのネットワークから切り離されてしまっても，分断されたノードはリクエストを受信可能である．センサデータの保存に関してはすべてのリクエストを処理可能である．ただし，センサデータの参照は他のノードと通信できないためそのノードが保持しているデータに限る．

センサデータサーバにおいての独立したノードの振る舞いについて考察する．センサデータサーバは 2.2 節で述べたようにできるだけ多くのセンサデータを保存する必要がある．独立したノードがセンサデータを保存するという点に関してはそれぞれ保存することが可能である．ただし，分散 RDB に関しては独立したノードの IP アドレスがわかっている必要がある．システムが正常に動作している状態では SPDER ノードへのアクセスしか行わない．障害がおこった場合だけ独立したノードへリクエストを割り振るのは困難である．一方分散 KVS では，一カ所の IP アドレスにリクエストを送る必要がない．そのため，アプリケーションやセンサごとアクセスを各ノードに割り振っておくことでネットワーク障害時にも対応できる．

5.5 障害復旧時

ネットワークに障害が発生し，一部ノードが分散システムから切り離されてしまった後，ネットワークが復旧した場合についてデータがどうなるのか比較する．

分散 RDB の場合，分断される前のデータはそのまま使用可能である．しかし，ネットワークが分断されている間，そのノードに対するリクエストは処理されていない．そのため，ネットワークが分断されていた間のデータは記録されていない状態となる．独立した状態でノードに直接リクエストを送りデータを保存していても，システムには反映されない．このデータを反映させるには，そのノードのデータをすべてダンプし，SPIDER ノードを経由してリストアする必要がある．リストアには長時間を必要とする．

分散 KVS の場合，ノードが分断されてしまった場合もリクエストを受け付けデータを保存することが可能である．それに加え Riak は 4.1.2 節で述べたようにゴシッププロトコルで常にクラスタを監視している．システムに復帰したノードはゴシッププロトコルによって検出され，コンシステント・ハッシングに追加される．その際に障害前のデータと復旧後のデータで不整合が生じる．しかし，Riak はリード・リペアやヒンテッド・ハンドオフのような仕組みで整合性の修復が可能である．リード・リペアとはすべての複製データを比べ，もし差異があればバックグラウンドで修復するものである．ヒンテッド・ハンドオフは，複製を作成するノード B が稼働していない場合，別のノード C に複製を作成する．もしノード B が復旧した場合はデータを更新しておくようノード C に命令しておく方法である．これらの方法でデータの整合性を修復する際にはタイムスタンプやベクタークロックといった方法が使用される．

復旧後のデータ統合は分散 RDB も分散 KVS も可能である．ただし，分散 RDB はリストアに長時間かかることが予想され，復旧が分散 KVS に比べ高コストである．

6. おわりに

本稿では，センサデータサーバの実装に向けて，分散 RDB と分散 KVS についてデータ形式, スケーラビリティや可用性などについて考察した．考察を行うために SPIDER を用いて分散 RDB，Riak を用いて分散 KVS を仮想サーバ上に構築した．各環境を構築することや，パフォーマンスを計測することでそれぞれの比較を行った．

今後の課題に分散 KVS の複雑な検索の高速化，高速化のための KVS の設計や仮想上でなく物理的な分散環境で構築することなどが挙げられる．パフォーマンスを計測した結果，複雑な検索を行った場合分散 KVS は分散 RDB に比べ検索速度が遅く，実用的でなかった．KVS の設計や，MapReduce の活用をどのようにすると複雑な検索のパフォーマンスが向上するのか模索する必要がある．また，今回仮想環境上に複数のノードを作成して分散環境を構築した．一つのハードディスクに複数のノードが多数アクセスする状況が原因で，パフォーマンスが低下した可能性がある．物理的に分散環境を構築するか，仮想ノードごとに異なるハードディスクを割り当てるなどして再度パフォーマンスをテストすることも今後の課題である．

参考文献

[1] 馬場口登, 美濃導彦, ” 特集「センシングウェブ」にあたって”，人工知能学会誌，Vol.24，No.2，pp.177-178，2009.
[2] S. Ghemawat, H. Gobioff and S. T. Leung, ”The Google File System”, SOSP2003, pp.1-15, 2003.
[3] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall and W. Vogels, ”Dynamo: Amazon’s Highly Available Key-Value Store”, SOSP2007, pp.205-220, 2007.
[4] Hadoop,”http://hadoop.apache.org”
[5] 川上大輔，松井俊治，斉藤彰一，津邑公暁，松尾啓示，” 範囲検索と複数属性のデータの処理に適応した分散データストア”, IPSJ SIG Technical Report, Vol.2010-OS-113 No.10,pp1-9,2010.